# 付録B

コード

この付録は、ツールキットとして用いる階層化モナド変換子のためのSchemeのコードの一覧である。ただし、型変換と逆単位演算子（inverse unit operator）のためのコードは話を明解にするために除いた。

B.1 モナド変換子の定義

この節では最もよく見られるモナド変換子のコードを示す。

;; Environments: F(T)(A) = Env -> T(A)

(define (env-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a)

(lambda (env) (unit a)))

(lambda (c f)

(lambda (env)

(bind (c env)

(lambda (a)

((f a) env)))))

(lambda (c f)

(compute (c empty-env) f))

))))

図B.1：環境モナド変換子

;;; Exceptions: F(T)(A) = T(A + X)

(define (exception-trans t)

(with-monad t

(lambda(unit bind compute)

(make-monad

(lambda (a) (unit (in-left a)))

(lambda (c f)

(bind c (sum-function f (lambda (x) (unit (in-right x))))))

(lambda (c f)

(compute c (sum-function f compute-x)))

))))

図B.2：例外モナド変換子

;;; Continuations: F(T)(A) = (A -> T(Ans)) -> T(Ans)

(define (continuation-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a)

(lambda (k) (k a)))

(lambda (c f)

(lambda (k)

(c (lambda (a) ((f a) k)))))

(lambda (c f)

(compute (c (compose1 unit value->answer)) f))

))))

図B.3：継続モナド変換子

;;; Stores: F(T)(A) = Sto -> T(A \* Sto)

(define (store-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a)

(lambda (sto)

(unit (pair a sto))))

(lambda (c f)

(lambda (sto)

(bind (c sto)

(lambda (as)

((f (left as)) (right as))))))

(lambda (c f)

(compute (c (initial-store))

(lambda (a\*s)

(compute-store (f (left a\*s)) (right a\*s)))))

))))

図B.4：ストアモナド変換子

;;; Lifting 1: F(T)(A) = 1 -> T(A)

(define (lift1-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a)

(lambda () (unit a)))

(lambda (c f)

(lambda ()

(bind (c) (lambda (a) ((f a))))))

(lambda (c f)

(compute (c) f))

))))

図B.5：第一持ち上げ変換子

;;; Lifting 2: F(T)(A) = T(1 -> A)

(define (lift2-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a)

(unit (lambda () a)))

(lambda (c f)

(bind c (lambda (l) (f (l)))))

(lambda (c f)

(compute c (lambda (l) (f (l)))))

))))

図B.6第二持ち上げ変換子

;;; Lists: F(T)(A) = T(List(A))

(define (list-trans t)

(with-monad t

(lambda (unit bind compute)

(define (amb x y)

(bind x

(lambda (x)

(bind y

(lambda (y)

(unit (append x y)))))))

(make-monad

(lambda (a)

(unit (list a)))

(lambda (c f)

(bind c

(lambda (l)

(reduce amb (unit '()) (map f l)))))

(lambda (c f)

(compute c (lambda (l) (map f l))))

))))

図B.7：リストモナド変換子

;;; Monoids: F(T)(A) = T(A \* M)

(define (monoid-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a) (unit (pair a (monoid-unit))))

(lambda (c f)

(bind c

(lambda (a\*m)

(let ((c2 (f (left a\*m))))

(bind c2

(lambda (a\*m2)

(unit

(pair (left a\*m2

(monoid-product

(right a\*m) (right a\*m2)))))))))))

(lambda (c f)

(compute

c (lambda (a\*m)

(compute-m (f (left a\*m)) (right a\*m)))))

))))

図B.8：モノイドモナド変換子

;;; Resumptions: F(T)(A) = fix(X) T(A + X)

(define (resumption-trans t)

(with-monad t

(lambda (unit bind compute)

(make-monad

(lambda (a) (unit (in-left a)))

(lambda (c f)

(let loop ((c c))

(bind c

(sum-function

f (lambda (c)

(unit (in-right (loop c))))))))

(lambda (c f)

(compute

(let loop ((c c))

(bind c

(sum-function

(compose1 unit f)

loop)))

id))

))))

図B.9：再開機能モナド変換子